3DCrypt: Privacy-preserving Pre-classification Volume Ray-casting of 3D Images in the Cloud

Manoranjan Mohanty[1], Muhammad Rizwan Asghar[2] and Giovanni Russello[2]
[1]*Center for Cyber Security, New York University Abu Dhabi, Abu Dhabi, U.A.E.*
[2]*Department of Computer Science, The University of Auckland, Auckland, New Zealand*

Keywords: Encrypted Volume, Paillier Cryptosystem, Cloud-based Secure Volume Ray-casting.

Abstract: With the evolution of cloud computing, organizations are outsourcing the storage and rendering of volume (*i.e.,* 3D data) to cloud servers. Data confidentiality at the third-party cloud provider, however, is one of the main challenges. In this paper, we address this challenge by proposing – *3DCrypt* – a modified Paillier cryptosystem scheme for multi-user settings that allows cloud datacenters to render the encrypted volume. The rendering technique we consider in this work is pre-classification volume ray-casting. *3DCrypt* is such that multiple users can render volumes without sharing any encryption keys. *3DCrypt*'s storage and computational overheads are approximately 66.3 MB and 27 seconds, respectively when rendering is performed on a $256 \times 256 \times 256$ volume for a 256×256 image space.

1 INTRODUCTION

Nowadays, cloud computing is increasingly used by organizations for rendering of volume data (*i.e.,* 3D data) (Cuervo et al., 2015; KDDI Inc., 2012; Intel Inc., 2011). Although this cloud-based rendering has many advantages, data confidentiality, which can lead to privacy, is one of the main concerns. For example, the data can be a scanned MRI of a patient's head, whose storage and rendering were outsourced to the cloud by a hospital. The data encryption scheme must be such that rendering can be performed on the encrypted data. Although the rendered image must not disclose any information to the cloud server, an authorized user (the one holding the encryption key) must be able to discover the secret rendered image from the encrypted rendered image.

Ideally, one would like to use the fully homomorphic encryption scheme to perform any type of computations over encrypted data (Baharon et al., 2013). However, current homomorphic encryption schemes are not computationally practical. Therefore, partial homomorphic schemes have been typically used for the cloud-based encrypted processing. Using the partial homomorphic Shamir's (k,n) secret sharing, Mohanty *et al.* (Mohanty et al., 2012) proposed the encrypted domain pre-classification volume ray-casting. Their work, however, cannot hide shape of the object from a datacenter and can disclose the color of the object when k or more datacenters collude together. Recently, Chou *et al.* (Chou and Yang, 2015) used permutation and adjusted the color transfer function (used in rendering) such that critical information about the object can be hidden from the rendering server. Their work, however, is not secure enough.

When working on encrypted data, usually a lot of attention is paid to the actual scheme without considering key management, an aspect critical for organizations. In a typical organization, issuing the same key to all employees, who want to share data, is not feasible. In an ideal situation, each employee must have her own personal key that can be used to access data encrypted by the key of any other employ. This scenario is often referred to as the full-fledged multi-user model. When the employee leaves the organization, the employee's key must be revoked and the employee must not be able to access any data (including her own data) anymore. However, the data must be accessible to employees still holding valid keys.

In this paper, we present *3DCrypt*, a cloud-based multi-user encrypted domain pre-classification volume ray-casting framework based on the modified Paillier cryptosystem. Paillier cryptosystem is homomorphic to only addition and scalar multiplication operations, whereas the pre-classification volume ray-casting performs a number of operations, including additions, scalar multiplications and multiplications. The use of Paillier cryptosystem alone there-

Mohanty, M., Asghar, M. and Russello, G.
3DCrypt: Privacy-preserving Pre-classification Volume Ray-casting of 3D Images in the Cloud.
DOI: 10.5220/0005966302830291
In *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications (ICETE 2016) - Volume 4: SECRYPT*, pages 283-291
ISBN: 978-989-758-196-0

fore cannot hide the shape of the object as the rendering of opacity, which basically renders the shape, involves multiplications. We addressed this issue using a private-public cloud model in such a way that rendering tasks can be distributed among the public and private cloud servers without disclosing both shape and color of the object to any of them.

Our contributions are summarized as follows:

- We provide a full-fledged multi-user scheme for the encrypted domain volume rendering.
- Unlike Mohanty *et al.*'s Shamir's secret sharing-based work, we can hide both color and shape of the object from the cloud.
- *3DCrypt* provides more security against collusion attacks than Mohanty *et al.*'s work.

Our preliminary analysis performed on a single machine shows that *3DCrypt* requires significant overhead. According to our analysis, the computation cost at the user-end, however, can be affordable.

For example, when rendering a $256 \times 256 \times 256$ for 256×256 image space, the computation overhead at the public cloud server and the private cloud server are 16.5 minutes and 15.2 minutes, respectively. For this data and image space, the data overhead and computation overhead at the image user-end are approximately 66.3 MB and 27 seconds, respectively.

2 RELATED WORK

There are very few works in the direction of encrypted domain rendering using the partial homomorphic encryption. Mohanty *et al.* proposed the encrypted pre-classification volume ray-casting (Mohanty et al., 2012) and the encrypted post-classification volume ray-casting (Mohanty et al., 2013) using Shamir's (k,n) secret sharing. Their pre-classification volume ray-casting scheme, however, cannot hide shape of the rendered object from the cloud server. Furthermore, their scheme requires n datacenters and assumes that k of them must never collude. Recently, Chou *et al.* (Chou and Yang, 2015) proposed a privacy-preserving volume ray-casting scheme that uses *block-based* permutation (which creates sub-volumes from the volume and permutes the sub-volumes) and adjustment of transfer function to hide both the volume and rendering tasks from the rendering server. Their work, however, lacks to achieve privacy due to the loss of information from the adjusted transfer table and the use of permutation.

In literature, there are alternative schemes to address privacy issues by outsourcing of volume rendering and visualization tasks to a third-party server. Koller *et al.* (Koller et al., 2004) protected the high-resolution data from an untrusted user by only allowing the user to view the low-resolution results during interactive exploration. Similarly, Dasgupta and Kosara (Dasgupta and Kosara, 2011) proposed to minimize the possible disclosure by combining non-sensitive information with high sensitivity information. The major issue with such schemes is that they leak sensitive information.

3 SYSTEM MODEL

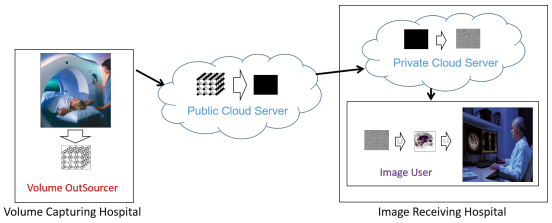


Figure 1: Cloud-based rendering of medical data.

In this work, we consider a distributed cloud-based rendering system, where a cloud server stores and renders volumes on behalf of a volume data outsourcer. Figure 1 presents a real-world medical imaging scenario of *3DCrypt*. In the system model, we assume the following entities.

- **Volume Outsourcer:** This entity outsources the storage and rendering of volumes to a third-party cloud provider. It could be an individual or part of an organization. In the latter case, users can act as Volume Outsourcers. Typically, this entity owns the volume. The Volume Outsourcer can store new volumes on a cloud server, delete/modify existing ones and manage access control policies (such as read/write access rights).In our scenario, the Volume Outsourcer is part of a volume capturing hospital.
- **Public Cloud Server:** A Public Cloud Server is part of the infrastructure provided by a cloud service provider, such as Amazon S3[1], for storing and rendering of volumes. It stores (encrypted) volumes and access policies used to regulate access to the volume and the rendered image. It performs most of the rendering on stored volumes and produces the partially rendered data.
- **Private Cloud Server:** The Private Cloud Server sits between the Public Cloud Server and the rendering requester. It can be part of the infrastructure, either provided by a private cloud service provider or maintained by an organization

[1]https://aws.amazon.com/s3/

as a proxy server. The Private Cloud Server receives partially rendered data from the Public Cloud Server and performs remaining rendering tasks on the volume. It then sends the rendered image to the rendering requester. Note that the Private Cloud Server does not store data, it only performs minimal rendering operations on partially rendered data received from the Public Cloud Server.

- **Image User:** This entity is authorized by the Volume Outsourcer to render a volume stored in the Public Cloud Server. In a multi-user setting, an Image User can (i) render an image (in encrypted domain) that will be accessible by other Image Users, or (ii) access images rendered by other Image Users. In both cases, Image Users do not need to share any keying material.
- **Key Management Authority (KMA):** The KMA generates and revokes keys to entities involved in the system. For each user (be a Volume Outsourcer or Image User), it generates a key pair containing the user-side key and the server-side key. The server-side key is securely transmitted to the Public Cloud Server, whereas, the user-side key is either sent to the user or Private Cloud Server depending on whether the user is a Volume Outsourcer or Image User. Whenever required (say in key lost or stolen cases), the KMA revokes the keys from the system with the support of the Public Cloud Server.

Threat Model: We assume that the KMA is a fully trusted entity under the control of the Volume Outsourcer's organization. Typically, the KMA can be directly controlled by the Volume Outsourcer. Since the KMA deals with small amount of data, it can be easily managed and secured. We also assume that the KMA securely communicates the key sets to the Public Cloud Server and the Image User. This can be achieved by establishing a secure channel. Except for managing keys, the KMA is not involved in any operations. Therefore it can be kept offline most of the times. This also minimizes the chances of being compromised by an external attack.

We consider an *honest-but-curious* Public Cloud Server. That is, the Public Cloud Server is trusted to honestly perform rendering operations as requested by the Image User. However, it is not trusted to guarantee data confidentiality. The adversary can be either an outsider or even an insider, such as unfaithful employees. Furthermore, we assume that there are mechanisms to deal with data integrity and availability of the Public Cloud Server.

In *3DCrypt*, the Private Cloud Server is an *honest-and-semi-trusted* entity. The Private Cloud Server is also expected to honestly perform its part of rendering operations. The Private Cloud Server is semi-trusted in the sense that it cannot analyze more than what can perceptually be learnt from the plaintext volume. We assume that the Private Cloud Server is in conflict of interest with the Public Cloud Server. That is, both cloud servers should not collude.

4 PROPOSED APPROACH

In this section, we present the architecture and an overview of work flow of *3DCrypt*.

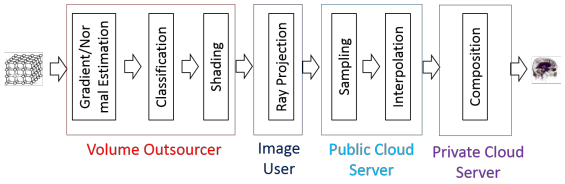


Figure 2: Distribution of rendering pipeline.

Figure 2 provides an overview of the pre-classification volume rendering pipeline, and illustrates how we distribute the rendering pipeline among different components of *3DCrypt*. The Volume Outsourcer performs *pre ray-projection rendering* operations: gradient/normal estimation, classification, and shading as these one-time operations can be pre-processed. The output of these operations are encrypted using Paillier cryptosystem, and the encrypted volume is sent to the Public Cloud Server. The Image User projects rays to the encrypted volume stored on the Public Cloud Server. The Public Cloud Server performs part of the *post ray-projection rendering* operations: sampling and interpolation on encrypted colors and opacities. Then, the Public Cloud Server sends interpolated colors and opacities to the Private Cloud Server. The Public Cloud Server, however, does not share information about voxel coordinates and projected rays with the Private Cloud Server. The Private Cloud Server decrypts interpolated opacities, and performs remaining post ray-projection rendering operations: color and opacity compostion using plaintext opacities and encrypted colors. Then, the Private Cloud Server sends the encrypted composite colors and plaintext composite opacities to the Image User. Finally, the Image User decrypts composite colors and creates the plaintext rendered image using plaintext colors and opacities.

Figure 3 illustrates the architecture of *3DCrypt*. In this system, the Volume Outsourcer is responsible for storing a volume and defining access policies for

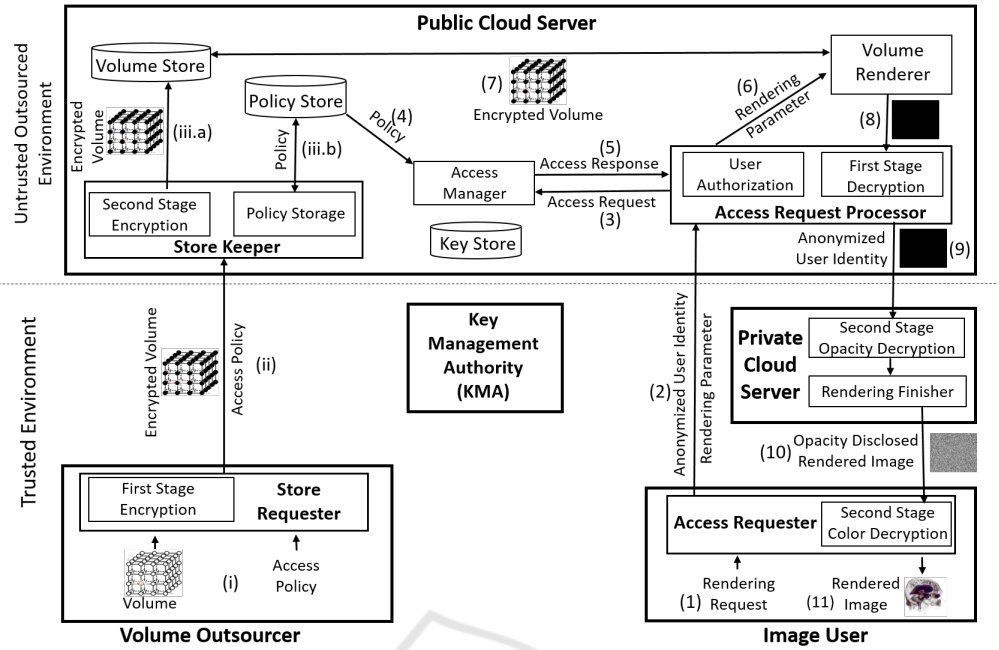


Figure 3: The architecture of *3DCrypt*, a cloud-based secure volume storage and rendering system.

the volume. To achieve this, the Volume Outsourcer interacts with the client module *Store Requester*. The Volume Outsourcer provides plaintext volume and access policies (Step i). The Store Requester performs the first stage encryption on the input volume using the user-side key and then sends the encrypted volume along with associated access policies to the *Store Keeper* module of the Public Cloud Server (Step ii). On the Public Cloud Server-end, the Store Keeper performs the second stage of encryption using the server-side key corresponding to the user and stores the encrypted volume in a *Volume Store* (Step iii.a). The Store Keeper also stores access policies of the volume in the *Policy Store* (Step iii.b).

Once an Image User expects the Public Cloud Server to render a volume, her client module *Access Requester* receives her rendering request in the form of projected rays (Step 1). The module *Access Requester*, in turn, forwards the request to the *Access Request Processor* module of the Public Cloud Server (Step 2). In the request, the Access Requester sends rendering parameters (such as direction of projected rays) and user credentials (which can be anonymized) to the Access Request Processor. The Access Request Processor first performs the user authorization check to find out if the user has authorization to perform requested operations. For this purpose, the *Access Manager* is requested for checking access policies (Step 3). The Access Manager fetches access policies from the Policy Store (Step 4). Next, it matches access policies against the access request. Then, the access response is sent back to the Access Request Processor (Step 5). If the user is authorized to perform the requested operation, the *Volume Renderer* is invoked with rendering parameters (Step 6). The requested volume is retrieved from the Volume Store (Step 7). Then, most of the rendering tasks, which do not require multiplication of opacities, are performed in the encrypted manner and the partially rendered data is sent to the Access Request Processor (Step 8). The Access Request Processor performs the first round of decryption on the rendered data, hides voxel positions (*e.g.,* permuting coordinates of the voxels) and sends the data, protected pixel positions and the user identity to the Private Cloud Server (Step 9). The Private Cloud Server performs the second round of decryption and obtains partially rendered opacities in plaintext. The plaintext opacities and encrypted colors are sent to the *Rendering Finisher* module, which performs rest of rendering tasks involving multiplication of opacities. Since the opacities are in plaintext, the multiplication of opacities with colors are reduced to a scalar multiplication. The Private Cloud Server then sends the opacity disclosed (but shape protected as voxel positions are unknown) and color encrypted rendered image to the Access Requester (Step 10). The Access Requester decrypts the colors and shows the rendered image to the Image User (Step 11).

5 SOLUTION DETAILS

3DCrypt is based on a modified Pailler cryptosystem that supports re-encryption (Bresson et al., 2003; Ateniese et al., 2006; Ayday et al., 2013) and is homomorphic to additions and scalar multiplications. Therefore, using this cryptosystem, we can encrypt a volume, for which rendering has been adjusted such that the post ray-projection rendering tasks including interpolation and composition can be performed by a combination of additions and scalar multiplications. In order to provide the multi-user support, we extend the modified Paillier cryptosystem such that each user has her own key to encrypt or decrypt the images. Therefore, for adding a new user or removing an existing one, re-encryption is not required.

The main goal of *3DCrypt* is to leverage resources of the Public Cloud Server as much as possible by storing the data volume and by performing most of the rendering tasks. To ensure confidentiality, both colors and opacities are stored in an encrypted form. Information about projected rays, however, must be available in plaintext as it is required in sampling and interpolation steps of the rendering operation. Since the Paillier cryptosystem is non-homomorphic to multiplication, composition, which multiplies opacities, cannot be computed on encrypted opacities. Thus, we employ a private cloud that can perform composition by knowing plaintext opacities. Since the knowledge of opacities and pixel positions can perceptually disclose shape of the rendered image, we consider to protect voxel positions from the Private Cloud Server. We can achieve this by permuting voxel positions to dissociate them from projected rays.

A key difficulty in integrating the Paillier cryptosystem with volume ray-casting is the incompatibility of floating point operations of ray-casting operations with the modular prime operation of the Paillier cryptosystem. One way of overcoming this issue is by converting the floating point interpolating factors and opacities to their fixed point representatives. We can achieve this by rounding off a float by d decimal points and multiplying 10^d with the round-off number.

Figure 4 provides the technical overview of *3DCrypt*'s rendering system. In *3DCrypt*, the KMA generates the color-key-set $\{K_S^C, K_U^C\}$ and the opacity-key-set $\{K_S^A, K_U^A\}$ for each user in the system (either acting as a Volume Outsourcer or Image User). Each key set contains pair of keys: the user-side key and the server-side key. For each user i, the server-side key of the color-key-set and the opacity-key-set, which are $K_{S_i}^C$ and $K_{S_i}^A$ respectively, are securely transmitted to the Public Cloud Server. The Public Cloud Server securely stores all the server-side keys in the *Key Store*, which could be accessible only to the Store Keeper and the Access Request Processor. When the user i is the Volume Outsourcer, the user-side keys of the color-key-set and the opacity-key-set: $K_{U_i}^C$ and $K_{U_i}^A$, are transmitted to the user. However, when the user i is the Image User, then the user-side key of color-key-set and opacity-key-set, $K_{U_i}^C$ and $K_{U_i}^A$, are sent to the user and the Private Cloud Server, respectively.

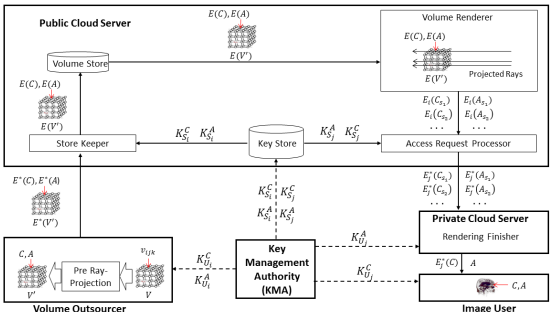


Figure 4: Encryption and decryption processes using *3DCrypt*.

The workflow of our secure rendering system can be divided into three major steps: data preparation, ray-dependent rendering and composition.

5.1 Data Preparation

This step is performed by the Volume Outsourcer prior to projection of rays. In this step, the Volume Outsourcer performs two main tasks: (i) pre ray-projection rendering and (ii) encryption of output of the pre ray-projection rendering using user-side keys. The pre ray-projection rendering maps the physical property v_{ijk} of the ijk^{th} data voxel to its corresponding color C and opacity A values. After this step, an input volume V is represented as V', where the ijk^{th} voxel of V' contains colors and opacity found by the physical property of the ijk^{th} voxel of V.

For a user i, the colors and opacities of V' are encrypted using $K_{U_i}^C$ and $K_{U_i}^A$, respectively (see Section 5.4. The encryption outputs $E_i^*(C)$ and $E_i^*(A)$ are sent to the Store Keeper as an encrypted volume $E_i^*(V')$. The Store Keeper then uses the server-side keys $K_{S_i}^C$ and $K_{S_i}^A$ to re-encrypt $E_i^*(C)$ and $E_i^*(A)$ and stores the re-encrypted volume $E(V')$ in the Volume Store.

In the encryption process, we adopt two main optimizations to decrease the storage overhead. First, we use an optimized modified Paillier cryptosystem. Second, we encrypt three color components – R, G and B – in a single big number rather than encrypting them independently. We discuss both the optimizations below.

Optimization of the Modified Paillier Cryptosystem. The modified Paillier cryptosystem is represented as: $E(C) = (e_1, e_2)$, where $e_1 = g^r$ and $e_2 = g^{rx} * (1 + Cn)$, and C is the plaintext color. Likewise, we encrypt the opacity A. Note that e_1 is independent of the plaintext color. By using a different e_1 for a different color (a typical case), we need $2k$ bits (where k is a security parameter) for storing e_1 of each color. We propose to optimize this space requirement by using one e_1 for encrypting t colors, requiring $2k$ bits for storing e_1 for all t colors. This optimization can be achieved by using the same random number r for all t colors.

Encrypting Color Components. Three color components (*i.e.,* R, G and B) undergo the same rendering operations. Each color component requires 8 bits in plaintext, but is represented by $2 * k$ bits when encrypted independently. We can reduce this overhead by representing three color components as a single big number and encrypting this number in the place of encrypting the color components independently. This encrypted number will then undergo rendering in the place of rendering of color components. One trick to create a big number from color components is by multiplying $10^{3*m*(d+f)}$ (where $d+f$ represents total rounding places during rounding operations in rendering) to m^{th} color component and add all the multiplications.

5.2 Ray-dependent Rendering

In this step, the Public Cloud Server first fetches encrypted volume $E(V')$ from the Volume Store and then performs sampling and interpolation on $E(V')$. We use the same sampling technique as used by the conventional ray-casting. The interpolation, however, is performed on the encrypted color $E(C)$ and opacity $E(A)$. The interpolation can be performed as additions and scalar multiplications when the interpolating factor I_{ijk} is available in plaintext. We therefore disclose I_{ijk} to the Public Cloud Server. Since the floating point I_{ijk} cannot be used with encrypted numbers, the Public Cloud Server converts I_{ijk} to an integer I'_{ijk} by first rounding-off I_{ijk} to d decimal places and then multiplying 10^d to the round-off value. *3DCrypt* is such that it allows the Public Cloud Server to run additions and scalar multiplications over encrypted numbers, as shown in Equations 1 and 2, respectively.

$$E(C_1) * E(C_2) = E(C_1 + C_2) \tag{1}$$

$$E(C)^{I'_{ijk}} = E(I'_{ijk} * C) \tag{2}$$

Likewise, we can compute opacity in an encrypted manner. The interpolated color $E(C_s)$ and opacity $E(A_s)$ for each sample point s are pre-decrypted using the Image User j's server-side keys $K^C_{S_j}$ and $K^A_{S_j}$, respectively. The pre-decrypted color $E^*_j(C_s)$ and opacity $E^*_j(A_s)$ are then sent to the Private Cloud Server along with the proxy ray to which the sampling point s is associated.

5.3 Composition

In this step, the Private Cloud Server accumulates the colors and opacities of all the sampling points along a proxy voxel position. Since this step involves multiplication of opacities (which are non-homomorphic to Paillier Cryptosystem), the Private Cloud Server performs the second round of decryption on $E^*_j(A_s)$ using the user j's user-side key for opacity, $K^A_{U_j}$. The multiplied opacities O_s, which will be multiplied with encrypted color, however is a floating point number. As discussed above, we can convert this float to an integer by first rounding-off the float by f places and then multiplying 10^f with the rounded-off value. Then, we can perform encrypted domain color composition using Equations 1 and 2. Note that since the available interpolated colors are in encrypted form, the composited color $E^*_j(C)$ also remains encrypted. Furthermore, in absence of voxel positions of the image space, the composited plaintext opacity A does not reveal shape of the rendered image.

The plaintext rendered opacity A and encrypted rendered color $E^*_j(C)$ are sent to the Image User, who decrypts $E^*_j(C)$ using $K^C_{U_j}$ and views the plaintext rendered image.

5.4 Construction Details

In this section, we described the algorithms used in our proposed scheme. We instantiate two instances of the scheme: one for the color while other for the opacity.

- **Init**(1^k)**.** The KMA runs the initialization algorithm in order to generate public parameters *Params* and a master secret key set *MSK*. It takes as input a security parameter k, and generates two prime numbers p and q of bit-length k. It computes $n = pq$. The secret key is $x \in [1, n^2/2]$. The g is of order: $\frac{\phi(n)}{2} = \frac{\phi(p)\phi(q)}{2} = \frac{(p-1)(q-1)}{2}$ and can be easily found by choosing a random $a \in \mathbb{Z}^*_{n^2}$ and computing $g = -a^{2n}$. It returns $Params = (n, g)$ and $MSK = x$. K_S represents the Key Store, initialised as $K_S \leftarrow \phi$.

- **KeyGen(MSK, i).** The KMA runs the key generation algorithm to generate keying material for users in the system. For each user i, this algorithm generates two key sets K_{U_i} and K_{S_i} by choosing a random x_{i1} from $[1, n^2/2]$. Then it calculates $x_{i2} = x - x_{i1}$, and transmits $K_{U_i} = x_{i1}$ and $K_{S_i} = (i, x_{i2})$ securely to user i and to the server, respectively. The server adds K_{S_i} to the Key Store as follows: $K_S \leftarrow K_S \cup K_{S_i}$.
- **ClientEnc(D, K_{U_i}).** A user i runs the data encryption algorithm to encrypt the data D using her key K_{U_i}. To encrypt the data $D \in \mathbb{Z}_n$, the user client chooses a random $r \in [1, n/4]$. It computes $E_i^*(D) = (\hat{e}_1, \hat{e}_2)$, where

$$\begin{aligned} \hat{e}_1 &= g^r \bmod n^2 \text{ and} \\ \hat{e}_2 &= \hat{e}_1^{x_{i1}}.(1+Dn) \bmod n^2 \\ &= g^{rx_{i1}}.(1+Dn) \bmod n^2. \end{aligned}$$

- **ServerReEnc($E_i^*(D)$, K_{S_i}).** The server re-encrypts the user encrypted data $E_i^*(D) = (\hat{e}_1, \hat{e}_2)$. It retrieves the key K_{S_i} corresponding to the user i and computes the re-encrypted ciphertext $E(D) = (e_1, e_2)$, where

$$\begin{aligned} e_1 &= \hat{e}_1 = g^r \bmod n^2 \text{ and} \\ e_2 &= \hat{e}_1^{x_{i2}}.\hat{e}_2 = g^{rx}.(1+Dn). \bmod n^2 \end{aligned}$$

- **ServerSum($E(D_1), E(D_2)$).** Given two encrypted values $E(D_1) = (e_{11}, e_{12})$ (where $e_{11} = g^{r_1}$ and $e_{12} = g^{r_1 x}.(1 + D_1 n)$ and $E(D_2) = (e_{21}, e_{22})$ (where $e_{21} = g^{r_2}$ and $e_{22} = g^{r_2 x}.(1 + D_2 n)$), the server calculates the encrypted sum $E(D_1 + D_2) = (e_1, e_2)$, where

$$\begin{aligned} e_1 &= e_{11}.e_{21} = g^{r_1 + r_2} \bmod n^2 \text{ and} \\ e_2 &= e_{12}.e_{22} \bmod n^2 \\ &= g^{(r_1+r_2)x}.(1 + (D_1 + D_2)n) \bmod n^2. \end{aligned}$$

- **ServerScalMul(c, $E(D)$).** Given a constant scalar factor c and an encrypted value $E(D) = (e_1, e_2)$ where $e_1 = g^r$ and $e_2 = g^{rx}.(1 + Dn)$, the server calculates the encrypted scalar multiplication $E(c.D) = (e_1^*, e_2^*)$, where

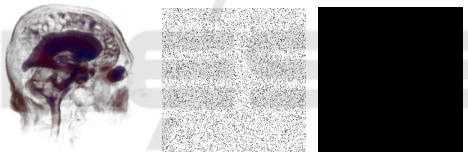
$$\begin{aligned} e_1^* &= e_1^c = g^{rc} \bmod n^2 \text{ and} \\ e_2^* &= e_2^c = g^{rcx}.(1 + cDn) \bmod n^2. \end{aligned}$$

- **ServerPreDec($E(D)$, K_{S_j}).** The server runs this algorithm to partially decrypt the encrypted data for the user j. It takes as input the encrypted value $E(D) = (e_1, e_2)$, where $e_1 = g^r$ and $e_2 = g^{rx}.(1 + Dn)$. The server retrieves the key K_{S_j} corresponding to the user j and computes the pre-decrypted data $E_j^*(D) = (\hat{e}_1, \hat{e}_2)$, where

$$\begin{aligned} \hat{e}_1 &= e_1 = g^r \bmod n^2 \text{ and} \\ \hat{e}_2 &= e_1^{-x_{j2}}.e_2 \bmod n^2 \\ &= g^{rx_{j1}}.(1 + Dn) \bmod n^2. \end{aligned}$$

- **UserDec($E_j^*(D)$, K_{U_j}).** The user runs this algorithm to decrypt the data. It takes as input the pre-decrypted data $E_j^*(D) = (\hat{e}_1, \hat{e}_2)$ where $\hat{e}_1 = g^r$ and $\hat{e}_2 = g^{rx_{j1}}.(1 + Dn))$, and her key K_{U_j}, and retrieves the data by computing: $D = L(\hat{e}_2.\hat{e}_1^{-x_{j1}}) = L(1 + Dn)$, where $L(u) = \frac{u-1}{n}$ for all $u \in \{u < n^2 | u = 1 \bmod n\}$.
- **Revoke(i).** The server runs this algorithm to revoke user i access to the data. Given the user i, the server removes K_{S_i} from the Key Store as follows: $K_S \leftarrow K_S \backslash K_{S_i}$.

6 IMPLEMENTATION AND EXPERIMENT



(a) (b) (c)

Figure 5: Secure rendering for the *Head* image. Figure 5a, Figure 5b and Figure 5c illustrate the rendered image available to the Image User, the Public Cloud Server and the Private Cloud Server, respectively.

We implemented the secure ray-casting by integrating the modified Paillier cryptosystem to the volume ray-casting module of the open source 3D visualization software VTK6.3.0. We run the implemented *3DCrypt* on a PC powered by Intel i5-4670 3.40 GHz processor and 8 GB of RAM, running Ubuntu 15.04. All the components of *3DCrypt*, *i.e.,* the Volume Outsourcer, the Public Cloud Server, the Private Cloud Server, the Image User and the KMA were simulated. Note that VTK is typically shipped with post-classification volume ray-casting. We modified VTK to provide pre-classification volume ray-casting. For dealing with big number cryptographic primitive operations, we integrated the MIRACL cryptographic library with VTK.

In our implementation, we chose a 1024-bit key size. We round-off the floating point numbers used in rendering operations by machine precision to avoid round-off errors. For the modified Paillier encryption,

we choose one random number r for all voxels of a volume, requiring one $e_1 = g^r$ (first cipher component) for all voxels.

Figure 5 illustrates how *3DCrypt* provides perceptual security in the cloud. An image available to the Public Cloud Server is all black since the Public Cloud Server does not know the color and opacity of the pixels. The image available to the Private Cloud Server, however, contains opacity information, which can disclose shape of the image as voxel positions are disclosed to the Private Cloud Server.

Performance Analysis. In *3DCrypt*, processing by the Volume Outsourcer and the encryption by the Public Cloud Server are one-time operations, which could be performed *offline*. The overheads of these operations, however, are directly proportional to the volume size. The overhead for a volume is equal to the product of a voxel's overhead with the total number of voxels in the volume (*i.e.,* the dimension of the volume). In our implementation, we need approximately 4064 bits more space to store the encrypted color and the opacity of a voxel (as two encryptions of 1024 bits key size are required for encrypting 32 bits RGBA values). Thus, we require approximately 8.6 GB of space to store a $256 \times 256 \times 256$ volume in encrypted domain (size of this volume in plaintext is approximately 67 MB). Similarly, for encrypting color and opacity of a voxel, the Volume Outsourcer requires approximately 540 millisecond (ms). The Public Cloud Server requires approximately 294 ms more computation with respect to the conventional plaintext domain pre-classification volume ray-casting implemented on the same machine. Thus, the Volume Outsourcer and the Public Cloud Server require approximately 2.52 hours and 1.37 hours, respectively, for encrypting the $256 \times 256 \times 256$ volume.

The rendering by the cloud servers and the decryption by the Image User are performed at runtime, according to the ray projected by the Image User. The overhead of performing these operations affects visualization latency, which is discussed below.

In *3DCrypt*, the overhead of transferring and performing the last round rendering operations in the Private Cloud Server is equal to the product of total number of sample points with the overhead of a sample point. Total number of sample points is equal to the sum of the sample points along all the projected rays and the number of sample points along a ray is implementation dependent. For rendering and decrypting (the first round) the color and opacity of a sample point, the Public Cloud Server requires approximately 290 ms of extra computation. For rendering and decrypting (the second round) opacity of a sample point, the Private Cloud Server requires approximately 265 ms of extra computation (with respect to the conventional plaintext domain pre-classification volume ray-casting).

In our implementation, for rendering and decrypting the $256 \times 256 \times 256$ volume data for a 256×256 image project space, the Public Cloud Server and the Private Cloud Server require approximately 16.5 extra minutes and 15.2 extra minutes, respectively. Note that for this data and image space, the data overhead at the Private Cloud Server is approximately 1.75 GB.

The overhead of transferring and decrypting the color-encrypted rendered image to the Image User is equal to the product of the number of pixels in the image space (which is equal to the number of projected rays) with the overhead for a single pixel. In *3DCrypt*, the Private Cloud Server must send approximately 2024 bits more data per pixel to the Image User. Therefore, for rendering a 256×256 image, the Image User must download 66.3 MB of more data than the conventional plaintext domain rendering. In addition, the Image User needs approximately 408 ms of computation to decrypt and recover rendered color of a pixel. Therefore, before viewing the 256×256 image, the Image User must work approximately 27 extra seconds.

Note that these results have been collected using a non-optimized version of our code. Our current implementation is based on MIRACL, which is also not optimized. Finally, to improve the performance of *3DCrypt*, we are currently working on a new parallel version to take advantage of a more realistic top-of-the-line hardware configuration used for the server-side.

7 CONCLUSIONS

Cloud-based volume rendering presents the data confidentiality issue that can lead to privacy loss. In this paper, we addressed this issue by encrypting the volume using the modified Paillier cryptosystem such that a pre-classification volume ray-casting can be performed at the cloud server in the encrypted domain. Our proposal, *3DCrypt*, provides several improvements over state-of-the-art techniques. First, we are able to hide both color and shape of the rendering object from a cloud server. Second, we provide better security to collusion attack than the state-of-the-art Shamir's secret sharing-based scheme. Third, users do not need to share keys for rendering volume stored in the cloud (therefore, maintenance of per-volume keys is not required).

To make *3DCrypt* more practical, a future work

can focus on decreasing the overheads both at the cloud and the user ends. Furthermore, it will be also be interesting to investigate whether we can extend *3DCrypt* for the encrypted domain post-classification volume ray-casting.

ACKNOWLEDGEMENT

This research is supported by STRATUS (Security Technologies Returning Accountability, Trust and User-Centric Services in the Cloud, https://stratus.org.nz), a project funded by the Ministry of Business, Innovation and Employment (MBIE), New Zealand.

REFERENCES

Ateniese, G., Fu, K., Green, M., and Hohenberger, S. (2006). Improved proxy re-encryption schemes with applications to secure distributed storage. *ACM TISSEC*, 9:1–30.

Ayday, E., Raisaro, J. L., Hubaux, J.-P., and Rougemont, J. (2013). Protecting and evaluating genomic privacy in medical tests and personalized medicine. In *ACM WPES*, pages 95–106.

Baharon, M., Shi, Q., Llewellyn-Jones, D., and Merabti, M. (2013). Secure rendering process in cloud computing. In *PST*, pages 82–87.

Bresson, E., Catalano, D., and Pointcheval, D. (2003). A simple public-key cryptosystem with a double trapdoor decryption mechanism and its applications. In *ASIACRYPT*, volume 2894 of *Lecture Notes in Computer Science*, pages 37–54.

Chou, J.-K. and Yang, C.-K. (2015). Obfuscated volume rendering. *The Visual Computer*, pages 1–12.

Cuervo, E., Wolman, A., and et al., L. P. C. (2015). Kahawai: High-quality mobile gaming using GPU offload. In *MobiSys*, pages 121–135.

Dasgupta, A. and Kosara, R. (2011). Adaptive privacy-preserving visualization using parallel coordinates. *IEEE TVCG*, 17(12):2241–2248.

Intel Inc. (2011). Experimental cloud-based ray tracing using intel mic architecture for highly parallel visual processing. Online Report.

KDDI Inc. (2012). Medical real-time 3d imaging solution. Online Report.

Koller, D., Turitzin, M., and et al., M. L. (2004). Protected interactive 3D graphics via remote rendering. In *ACM SIGGRAPH*, pages 695–703.

Mohanty, M., Atrey, P. K., and Ooi, W. T. (2012). Secure cloud-based medical data visualization. In *ACMMM*, pages 1105–1108, Nara, Japan.

Mohanty, M., Ooi, W. T., and Atrey, P. K. (2013). Secure cloud-based volume ray-casting. In *IEEE CloudCom*, Bristol, UK.